Attribute specifications for graphical interface generation

Paul Franchi-Zannettacci

INRIA

UNITE DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tel (1) 39 63 55 11

Rapports de Recherche

N°937

Programme 1

ATTRIBUTE SPECIFICATIONS FOR GRAPHICAL INTERFACE GENERATION

Paul FRANCHI-ZANNETTACCI

Décembre 1988

RR_0937

2935

ATTRIBUTE SPECIFICATIONS FOR GRAPHICAL INTERFACE GENERATION [1]

Paul FRANCHI-ZANNETTACCI [2]
LISAN – University of Nice
Avenue A. Einstein
06561 Valbonne Cedex, France
e-mail: pfz@cerisi.cerisi.fr

Keywords

Formal specifications, abstract syntax, attribute grammars, language-based editor, programs generator, programming environments, interactive graphics.

Abstract

This work deals with the automatic generation of graphic language-based editors. From an abstract syntax and a formal specification for the edition of graphical objects, we show how to generate an incremental editor for the involved objects supported by a bitmap workstation. The intended products are convivial programming environments like browsers, structured text-editors, graphical software, etc. The main features of this work are the use of (i) attribute grammars (AG) for graphic formal specification, (ii) a meta-language to generate AGs on nested boxes and (iii) incremental re-evaluation of the layout rules. Using well-known AG results, we prove that under acceptable conditions while using the meta-language, the generated AG belongs to the OAG class which provides the user with security and efficiency.

[1] This work is supported by CNRS-GRECO in Programming

[2] Currently at INRIA Sophia-Antipolis

SPECIFICATIONS ATTRIBUEES POUR LA GENERATION D'EDITEURS GRAPHIQUES [3]

Paul FRANCHI-ZANNETTACCI [4]
LISAN – Universitè de Nice
Avenue A. Einstein
06561 Valbonne Cedex, France
e-mail: pfz@cerisi.cerisi.fr

Mots clés

Spécifications formelles,syntaxe abstraite, grammaires attribuées, éditeur syntaxique, environnement de programmation, graphique interactif.

Résumé

Ce travail concerne la génération automatique d'éditeurs syntaxiques graphiques. A partir d'une spécification formelle sur une syntaxe abstraite, pour l'édition d'objets graphiques, nous montrons comment générer un éditeur incrémental pour ces objets sur une station "bitmap". Les domaines d'applications incluent les environnements de programmation conviviaux(Browser,...), les éditeurs de textes structurés, les logiciels graphiques interactifs, etc. Les points clés de ce travail sont (i) l'utilisation des grammaires attribuées (AG) pour la définition de spécifications graphiques, (ii) un méta-langage pour générer ces AGs sur un système de boites et (iii) une ré-évaluation incrémentale des règles d'affichage. En utilisant des rèsultats connus des AGs, nous prouvons que, sous des conditions raisonables, la grammaire attribuée obtenue à partir du méta-langage, appartient à la classe OAG, ce qui confère au système sécurité et efficacité.

[3] Cette étude est financée avec le concours du GRECO de Programmation

[4] Actuellement à l' INRIA Sophia-Antipolis

Contents

1 Executable specifications for graphical editors generation

This research is devoted to the language-based edition of tree-structured graphics and to the formal executable specifications from which such editors can be automatically generated. Extending the scope of well-known program or text manipulation systems such as MENTOR [9], CENTAUR [4], SYNTHESIZER GENERATOR [24], GANDALF [12], which do not support graphic editing, we want to provide the end-user with a convivial environment for editing both graphical and textual structured objects in a syntax directed way.

The development of such environments is a real challenge for the state of the art in software engineering. To be useful tools, indeed, these environments must offer very sophisticated functions and good performances while accepting continuous changes according to the specific needs of the user.

Our approach is based on a graphic specification language [20]. This language must be powerful enough to describe low-level aspects of layout, operational enough to be an input to a generator for producing an editor and friendly enough for a designer. This editor works on a bitmap workstation with interactive functionalities as windows, menus, icons and selection. The editing process within this environment is done accordingly to the rules involved in the generation of the editor.

Our solution provides a two-level specification mecanism:

1. **Specification by Attribute Grammars (AG)** [18].
   This methodology has been intensively used for compiler-compiler development and other semantic applications. We use it here to specify rules for computing layout attributes (position, size, bitmap image,...) attached to an abstract representation of the objects (graphic or textual). The AG technique implies several advantages in this domain:

   - The power of expression allows to specify very low-level features for graphics and thus to generate sophisticated edition functions.
   - The specification, expressed by rules in a declarative way, is a set of syntax-oriented linear equations or directed constraints on objects. The algorithms for computing the values involved in these equations are not specified by the designer but can be deduced from the specification [16,7,8].
   - This technique is doubly incremental: during the development step (as any rule-based system) and during the run-time step if the attribute system belongs to some well-known classes of AG [10,14,22].
   - The attribute specification can be also used for generating semantic tools (associated values, analysis, checks,...) and this enhances the consistency between editing and semantic processing [26,2].

2. **Specification by a graphical high level language (GSL).**
   This language avoids the fastidious work of hand-definition by AG in generating attributed equations from high-level specifications based on the nested box paradigm [19]. The key-point for this second level language is that we give it a full semantic definition by AG which leads to the following consequences :

- A GSL compiler (from graphic box specifications to attributed equations) can be obtained by a bootstrapping step using the Generator itself [annex A].
- Each GSL specification leads to a consistent set of attributed rules, in the sens of non circular AG [theorem section 4.3].
- The complexity, in terms of tree-traversals, for (re-)evaluating layout attributes has an upper bound for any GSL specification [corollary 1 in 4.3].
- This architecture makes it possible to easiliy extend the GSL language or tailor it to specific cases, by modifying its semantics and bootstrapping it again.

These paradigms: the attribute specifications and the GSL language are the basis of the GIGAS system [5], our current implementation under UNIX 4.2 BSD for a graphical language-based environment generator. The architecture of the GIGAS system consists in:

GSL : The Specification Language used by a *designer* to define a graphic editor for a language L (L_GE), translated into a standard attribute grammar L_AG.

GENERATOR : An attribute-based generator (currently the Synthesizer Generator) which produces from an attribute grammar upon a concrete and an abstract syntax:

- an incremental syntactic analyser for L (L_SA),
- a tree-processor according to the given abstract syntax,
- an incremental attribute evaluator for L_AG (L_AV).

LANGUAGE-BASED EDITOR : The graphical editor including the evaluator L_AV and the analyser L_SA generated for L, usable by an *end-user* via the GAM.

GAM : The graphical abstract machine exchanges messages with the editor and executes the physical requests for graphical logical operations. The current implementation uses X_Window V10 [6].

The GAM's architecture and functionalities are not dicussed in this paper. We only refer to some features of our current system to illustrate the GSL potentiality.

2 Attribute Specifications for Graphics

We consider here AG as a low-level language for defining textual, graphic or semantic properties. We use the box paradigm introduced by Knuth [19].

2.1 A box system

A box is a 2-D rectangular object caracterised by its *height (h), depth (d), width (W)* computed from a given *base-line* accordingly to the following figure:

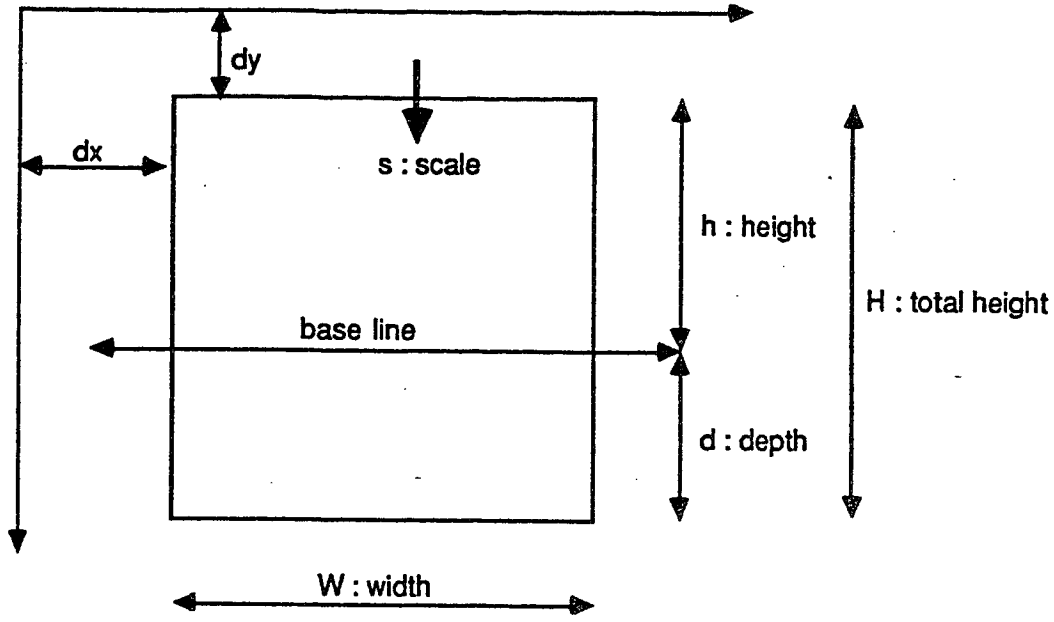


figure 1

We define a *vertical (dy)* and an *horizontal shift (dx)* w.r.t. a normal position defined by two reference axes (Ox,Oy). We use also a *scale (s)* for the final drawing.

2.2 Some nested boxes

Here are two examples issued from a formulae editor generated by GIGAS (annex D).

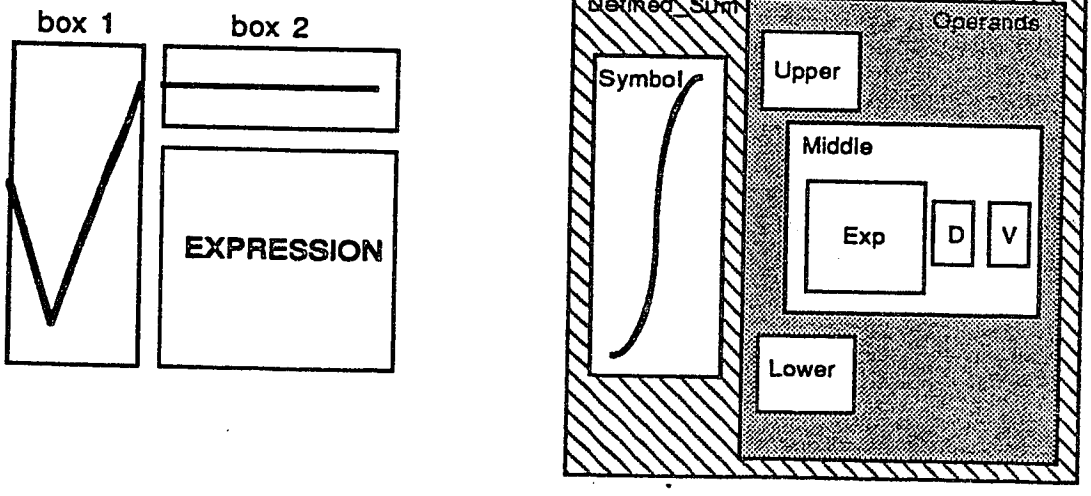


figure 2

2.3 An attribute system for boxes

The reader may find more than 600 references on AG in [8], we only recall that an attribute system consists in:

- **A concrete syntax** for the host language (in our case: defining the objects accepted by the editor).
- **An abstract syntax** for this language, given by a set of sorts and a set of tree-operators typed on these sorts.
- **A set of rules** to be computed (verified) on the objects in the language. These rules are equations on attribute-values which can be computed either locally w.r.t. an operator (*local* attributes), or bottom-up (*synthesized* attributes), or top-down (*inherited* attributes) in the abstract syntax tree.

An attribute system for specifying the layout of boxes consists in (see figure 2.1):

1. **a set of layout attributes** used for placing and drawing a box:

   H the total heigth of the box (local, H=h+d),

   W the width of the box (syn),

   X, Y the coordinates of the left-upper-corner relatively to the same corner of the father-box (inh).

2. **a set of auxiliary attributes** used for computing the boxes:

   s the scale (inh),

   h, d the heigth and depth w.r.t. the base-line (syn),

   dx, dy the horizontal and vertical shift w.r.t. a normal position (local by default).

   *As explained in the section 4.2, some of these attributes could be given default values such as dx = dy = 0 and h = d = 0.5 * s to obtain a standard display.*

3. **a set of equations** defining the rules for computing locally every attribute- value with two forms:

   (form1) BOX0.A = f(BOX0.Y, BOX1.B,...,BOXn.C)
   (form2) BOXk.Y = g(BOX0.Y, BOX1.B,...,BOXn.C)
   where

   BOX0 ⟶ BOX1,...,BOXn is an operator in the abstract syntax,
   A,B,C are synthesized or local attributes,
   Y is an inherited or local attribute,
   f,g are given fonctions well-typed with respect to attribute domains.

Remark 1 *The father-box forces the position and the scale (X, Y, s) of its sons (***top-down rules***) and its size (h, d, W) results from its sons (***bottom-up rules***). The X and Y attributes are values relative to the father-box; this solution is more efficient than absolute values for incremental re-evaluation and more convenient than relative to the previous box which implies an ordered layout.*

2.4 An example: the Defined_Sum operator

We give below the unparsing rules for the defined_sum expressions extracted from a usual mathematical formulae language.

1. a simplified concrete syntax is:

   DEFINED_SUM ::= **sum from** EXP1 **to** EXP2 **of** EXP3 **on** VAR **end sum**
   where

   EXP , VAR are non terminals described in the grammar,
   EXP1, EXP2 are limits of the sum,
   EXP3 the operand and VAR the differential variable.

2. the operator abstract syntax is given by the following tree-operator:

   defined_sum: EXP , EXP , EXP , VAR ⟶ EXP

   where EXP, VAR are sorts (phyla in MENTOR *terminology).*

3. a graphical display (shown in 2.2) is associated with a nested boxes system, which implies a box abstract syntax, defined by the following box-operators:

Defined_Sum:	SYMBOL, OPERANDS	⟶ DEFINED_SUM
Symbol:		⟶ SYMBOL
Operands:	UPPER, MIDDLE, LOWER	⟶ OPERANDS
Upper, Lower:	EXP	⟶ UPPER
Middle:	EXP, DIFF, VAR	⟶ MIDDLE
Diff:		⟶ DIFF

Where

the full capitalized words are box-phyla, the others are box-operators,
Symbol, Diff stand for pure graphic boxes,
Defined_Sum refers to the initial defined_sum operator,
the Exp, Var box-operators are omitted.

Remark 2 *From the point of view of attribute evaluation, we do not need to check the sorts of box-operators and consequently the pure box-operators are considered in*

the underlying one-sorted algebra (i.e. the pure box-operators have the same sort, say BOX, see section 4.1 for more details).

4. applied to defined_sum, we have the following equations (partial):

(top-down rules)

Symbol.s = Operands.s = Exp.s = Var.s = Defined_Sum.s
Upper.s = Lower.s = Defined_Sum.s * 0.8
Symbol.X = Defined_Sum.X + Symbol.dx
Operands.X = Defined_Sum.X + Symbol.W + Operands.dx

etc,

(bottom-up rules)

Defined_Sum.W = Symbol.W + Operands.W + Symbol.dx + Operands.dx
Defined_Sum.h = max(0,(Symbol.h-Symbol.dy),(Operands.h- Operands.dy))
Symbol.h = Operands.h - Upper.h

etc.

Remark 3 *These equations can be seen as a set of directed linear constraints between neighbouring boxes which express either general rules (horizontal and vertical nesting, alignment) or specific rules (decreasing scale, shift,...). Many equations can be replaced by default equations leading to a standard layout.*

Despite its numerous advantages (low-level declarative style, consistency decidability results, and generated incremental evaluators), AG technique needs a real effort in the case of full-sized development (for instance, more than 30,000 equations for an ADA source pretty-printer syntactic editor [6]). In the next part, we use previous remarks to define a high-level graphical language, fully compatible with AG to avoid redundancy of comparable descriptions in AG and therefore to notably decrease the charge of the designer.

3 The Graphical Specification Language GSL

In the case of large applications, we suggest to encapsulate AG by a more convenient language, GSL, a high-level specification language in the spirit of PPML [21]. GSL automatically produces attribute systems from box pre-defined types. Such a philosophy is enforced by AG methodoly: GSL is fully defined by semantic attributes and can be bootstrapped from its AG semantics (see annex A).

3.1 A first look at GSL

The GSL language is a first step towards a powerful specification language for graphics (see possible extensions in section 5). A GSL specification (or program) is a set of production rules according to a box abstract syntax and using typed boxes with a pre-defined layout. This layout can be tailored by user-defined equations on pre-defined box-attributes.

A production rule associates an abstract syntax operator to a box definition:

```
PRODUCTION      ::= LOCK phylum_IDENTIFIER : BOX ;
LOCK            ::= lock | unlock
```

where *LOCK regules the selection process for this production.*

A box definition refers to either an atomic or a composed box with a given type of layout (**horizontal, horizontal-centered, vertical, vertical-centered**).In both cases, equations may be provided to modify the standard layout:

```
BOX             ::= ATOMIC | COMPOSED
ATOMIC          ::= box_IDENTIFIER () EQUATIONS
                  | box_IDENTIFIER ( graphic string ) EQUATIONS
                  | box_IDENTIFIER ( string ) EQUATIONS
                  | box_IDENTIFIER (LOCK phylum_ IDENTIFIER ) EQUATIONS
COMPOSED        ::= box_IDENTIFIER TYPE ( BOX_LIST ) EQUATIONS
TYPE            ::= h | hc | v | vc
EQUATIONS       ::= EQUATION_LIST |
EQUATION        ::= ATTRIBUTE = expression ;
ATTRIBUTE       ::= dx | dy | h | d | l | s
```

where BOX_LIST,EQUATION_LIST, expression, string are omitted,
dx, dy, h, d, l, s are the box attributes introduced in section 2.3.

3.2 GSL source for the Defined_Sum expression

Let us comment a GSL source for *defined_sum* (we omit the meaning of the lock properties).

```
lock EXP:
    defined_sum hc (
- - a horizontal-centered layout for the defined_sum box ⟹ defined_sum's base line is the horizontal
- - reference axis of Symbol and Operands boxes, which are spread from left to right
                Symbol (graphic "sum")
                { h = Operands.h - Upper.h ;
                d = h ;
                W= (h + d) / 2.8 ; }
- - a box for the sum sign with a specific height and width
```

```
                Operandsv (
- - a vertical layout for the Operands box ⟹ the Upper, Middle and Lower boxes are vertically gathered
- - in this order along Operands's left border which is the vertical reference axis
                        Upper (unlock EXP ){ s = Symbol.s * 0.8 ; }
- - a reduced scale for the Upper and Lower boxes
                        Middle hc (
- - same base line for the Exp, Diff and Var boxes
                                Exp (unlock EXP)
                                Diff (" d")
                                Var (unlock VAR)
                        )
                        { dx = 1.0 ;
dy = max (Upper.d + Middle.h, Lower.h + Middle.h) + 1.0 * Middle.s - Upper.d - Middle.h ;
                        }
                        Lower (unlock EXP)
                        { s=Symbol.s * 0.8;
dy= max (Upper.d + Middle.h, Lower.h + Middle.h) + Middle.s - Middle.d - Lower.h;
                        }
- - a quite complex dy for the Lower box !
                        )
                {h=Upper.d + Upper.h + Middle.dy + Middle.h;}
- - a specific height for the Operands box
                );
```

Where DEFINED_SUM, EXP, VAR refer to the operator abstract syntax,
Symbol, Operands, etc refer to box abstract syntax,
defined_sum refer to both abstract syntaxes,
"d" is string to be printed and "sum" is a pre-defined graphic.

4 The theoretical framework of GSL

From a GSL specification on a box abstract syntax, a GSL compiler generates an AG over the underlying operator abstract syntax. The question answered in this section is the following: How can we know that the generated attribute system is or is not well-defined (non circular)? More than that does it necessary belong to one of the well-know optimized classes of AG (OAG, FNC,...) ? Beyond its theoretical interest, the answer to this question is the foundation of a pragmatic usage of this kind of techniques, to avoid inspecting the low-level generated AG for debugging GSL source.

4.1 From GSL to AG

Definition 1

Let P be a finite set of Phyla (Sorts); let O be a finite set of operators, defined as a P-signature, where each operator o has a result-type $\sigma(o)$ in P and an arity-type $\alpha(o)$ in P* (the free monoid on P); An abstract syntax on O, noted $\mathcal{A}(O)$ is the initial many sorted

algebra [1] generated by O.

Definition 2

A GSL specification G consists in:

1. An operator abstract syntax, $\mathcal{A}$(O).

2. A box abstract syntax, $\mathcal{A}$(B), built on B, a finite set of terms in $\mathcal{A}(O \cup I)$, where I denotes a set of Intermediate box-operators built on (P $\cup$ {BOX}), verifying for each i in I,
   $$\sigma(i) = BOX ;$$
   In addition, for each b in B the condition (1) holds:

   (1) b= $\hat{o}$ $(I_1, ..., I_n)$, with $I_i \in \mathcal{A}(I)$
   and for exactly one o in O:

   $name(\hat{o}) = name(o)$,
   $\sigma(\hat{o}) = \sigma(o)$,
   $\alpha(\hat{o}) = (BOX)^*$,
   $\alpha(b) = \alpha(o) = \alpha(I_1)...\alpha(I_n)$.
   where $\alpha(I_i)$ denotes the canonical extension of a from operators to trees.

3. A finite set of rules Γ_o, associated with O, defined as a set of equations on B, to compute pre-defined attributes on $\mathcal{A}$(B) with the form:

   bi.x = f (..., bi.y,.....) (pre-defined or user-defined equation)

   where bi, bj are in B, x,y are in { **dx, dy, h, l, d, s**} *and the whole equation follows the forms 1 or 2 in 2.3*

Property 1 *A GSL program defines an isomorphism between the operator and box abstract syntaxes.*

proof: direct consequence of the condition (1) in the definition.□

We use this isomorphism in both senses: from $\mathcal{A}$(O) to $\mathcal{A}$(B) for computing the visual display of objects as terms in $\mathcal{A}$(O) and conversely for evaluating in $\mathcal{A}$(O), graphic editing actions on $\mathcal{A}$(B) such as selection, insertion, etc.

Remark 4 *From a formal point of view, the generated equations are solved on terms in $\mathcal{A}(B)$, i.e. on terms in $\mathcal{A}(O \cup I)$. In practice, for efficiency reasons, we use local attributes, as defined in SSL [24], to compute attributes on I, locally to the associated operator in O, and so on terms in $\mathcal{A}(O)$ only. The GSL productions are primitive recursive schemes on $\mathcal{A}(O)$ [3].*

4.2 Equations generated by GSL

The set of equations generated from a GSL specification includes: default equations independant of the box type (general default equations), default equations associated with this type (design default equations) and user-defined equations , which, when present, are substituted to the default equations.

- **general default equations**

  for all boxes:
  [g1] BOXi.H = BOXi.h + BOXi.d
  [g2] BOXk.s = BOX0.s
  [g3] BOXi.dx = BOXi.dy = 0
  for $i \geq 0$ *and* $k > 0$;

  and for atomic boxes:
  [g4] BOX0.h = BOX0.d = 1/2 * BOX0.s
  [g5] BOX0.l = BOX0.d * length (BOX0)
  where length is a pre-defined fonction on atomic boxes

- **design default equations**

  We give here the equations generated by the h layout type as shown below:

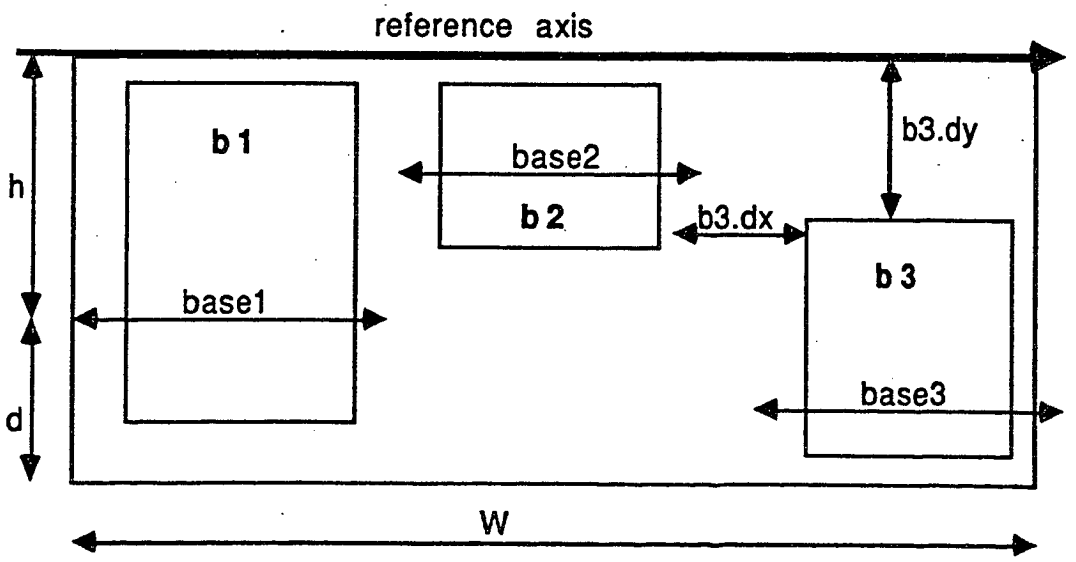


figure 3

Synthesized attributes of the father box:
[h1] BOX0.l = sum (BOXk.l + BOXk.dx)
[h2] BOX0.h = max (0 , max (BOXk.h - BOXk.dy))
[h3] BOX0.d = max (BOXk.h + BOXk.d + BOXk.dy) - BOX0.h
for k > 0 ;

Inherited attributes of the n sub-boxes:
[h4] BOX1.x = BOX1.dx + BOX0.x
" BOXj.x = BOXj-1.x + BOXj-1.l + BOXj.dx
[h5] BOXk.y = BOXk.dy + BOX0.y
for j > 1 and k > 0 ;

4.3 Fundamental properties

We use, without formal definiton, the following classic notations on AGs [28,8]:

- a<0> , a<k> for the values of an attribute a on a production p: $X_0 \longrightarrow X_1 ... X_k$
- $\mathcal{D}$(p), $\mathcal{D}$(S) the direct attribute depency graph over a production p or a sort S.
- $\mathcal{D}^*$(p), $\mathcal{D}^*$(S) the induced dependency graph over a production p or a sort S.
- ordered partitioned AG which determines two proper subclasses of non circular AG, the FNC (ANC) [17,7] and OAG [16] classes.

These classes are very relevant here: in addition to be tested by polynomial algorithms (instead of exponential ones in the general case [13]), they support incremental attribute evaluation [10] and value storage optimisations [22].

Let us analyse the direct dependencies between the attributes of the previous case on the assumption that there is no user-defined equation

from [g1]: $\mathcal{D}$(atomic_box) = (h<0>, H<0>), (d<0>, H<0>)
from [g2]: $\mathcal{D}$(atomic_box) = (s<0>, s<k>) , for k>0
from [g4] and [g5]: $\mathcal{D}$(atomic_box) = (s<0>, h<0>), (s<0>, d<0>), (s<0>, l<0>)
from [g1]: $\mathcal{D}$(h_box) = (h<0>, H<0>), (d<0>,H<0>)
from [g2]: $\mathcal{D}$(h_box) = (s<0>, s<k>) , for k>0
from [h1]: $\mathcal{D}$(h_box) = (l<k>, l<0>), (dx<k>, l<0>) , for k>0
from [h2]: $\mathcal{D}$(h_box) = (h<k>, h<0>), (dy<k>, h<0>) , for k>0
from [h3]: $\mathcal{D}$(h_box) = (d<k>, d<0>), (dy<k>, d<0>), (h<k>, d<0>), (h<0>, d<0>) , for k>0
from [h4]: $\mathcal{D}$(h_box) = (x<0>, x<1>), (dx<1>, x<1>) , (x<j-1>, x<j>), (dx<j>, x<j>), (l<j-1>, x<j>) , for j >1
from [h5]: $\mathcal{D}$(h_box) = (y<0>, y<k>), (dy<k>, y<k>) , for k>0

From the $\mathcal{D}$ graph, the OAG algorithm compute the induced dependencies graph $\mathcal{D}^*$

Lemma 1 *The AG generated from a set of GSL productions without user-defined equations including atomic and horizontal box specifications is OAG.*

Proof: This result is reached by computing $\mathcal{D}^*$ (BOX) and deducing an ordered partition of the set of attributes. We give here the partial order ($\longrightarrow$) resulting from $\mathcal{D}^*$ (h_box) and $\mathcal{D}^*$ (atomic_box).

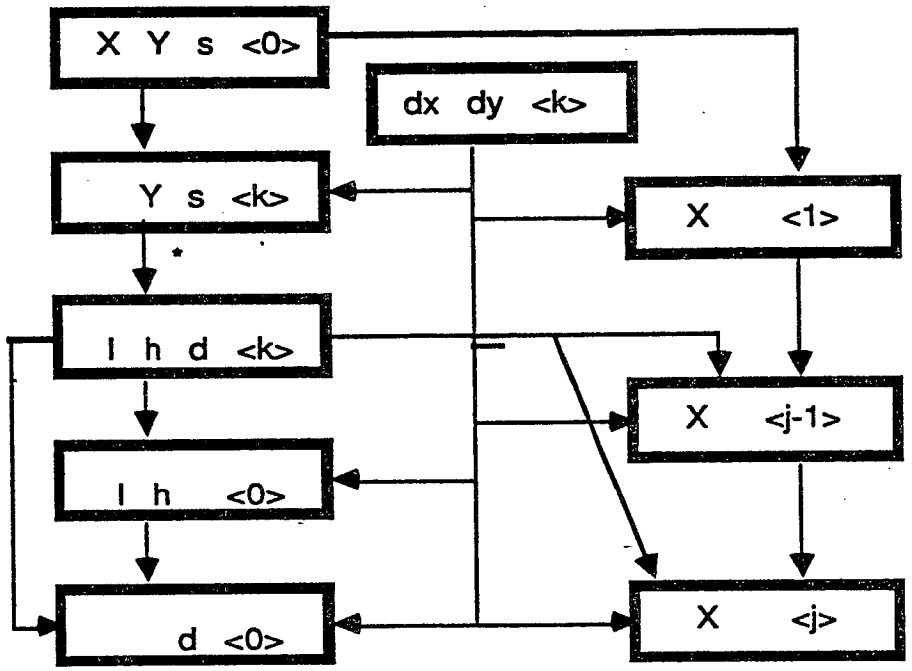


figure 4

An ordered partition of the attributes can be deduced from this diagram. From this partition, we deduce a total ordering of the attributes on a node BOX:
(h_BOX_ordering:) dx < dy < Y < s < l < h < d < X < H
which ends the proof of this lemma.□

We get the result which enforces security, efficency and convenience in our system.

Theorem 1 *The AG generated by a GSL program without user-equations is an OAG.*

Proof (sketch): we have three similar lemmas for the other cases: hc, v, vc which lead to four partial order H, HC, V, VC, for covering all GSL programs. We construct from these graphs a new partial order HVC compatible with the four previous ones. This partial

order does exist for GSL and supports the following total order
(BOX_ordering:) dx < dy < s <. l < h < d < H < X < Y □

This order gives an upper bound on the complexity for evaluating a GSL specification.

Corollary 1 *The evaluation of any GSL specification, without user-defined equations, costs at the most two tree-traversals.*

proof: as visible on the BOX ordering, we need to enter a BOX-node twice at the most; the first visit computes attributes from dx to H and the second visit, the attributes X,Y □

In the case where GSL specifications includes user-defined equations, the previous theorem is no longer true. In the general case, the designer must use well-know algorithms to test if the generated AG is circular, FNC or OAG. Deciding how extra equations modify the induced attribute dependencies sets the problem of incremental test for AG; it is out of the scope of this paper. However, we know an easy sufficient condition for keeping the OAG property of the generated system.

Corollary 2 *Let G be a GSL program with a set E of user-defined equations. If the direct dependencies issued from E are in the graph HVC from the previous theorem, then the generated AG from G is OAG.*

The proof is a consequence of the construction of the HVC graph.□

This result allows the designer to change the values and the functions on box attributes without any risk of circularity.

5 Conclusion and topics for future research

The design of the GIGAS system, as well as the different generated editors, point out two main advantages of our methodology:

1. Due to AG technology, we notabily decrease the time for producing a new editor and the designer is not asked to have any knowledge on graphical primitives;
2. The incremental evaluation gives the best performances during most editing actions.

The GSL specification can be improved in several ways:

- more standard features to decrease the role of user-defined equations,
- specifications over non-primitive abstract operators [21],
- extended attributes [11,23] for non strictly tree-structured graphics,
- non determistic features as multi-choice rules with conditional failure and logical graphic constraints on domain-variables [27].

Annex A: Implementation schema of GSL

We have used the generating power of the AG compiler-compiler technique (here SGEN: the Synthesizer Generator) to bootstrap the GSL translator from a definition of GSL by SSL.The next figure gives the detailed architecture of the GIGAS system [6].

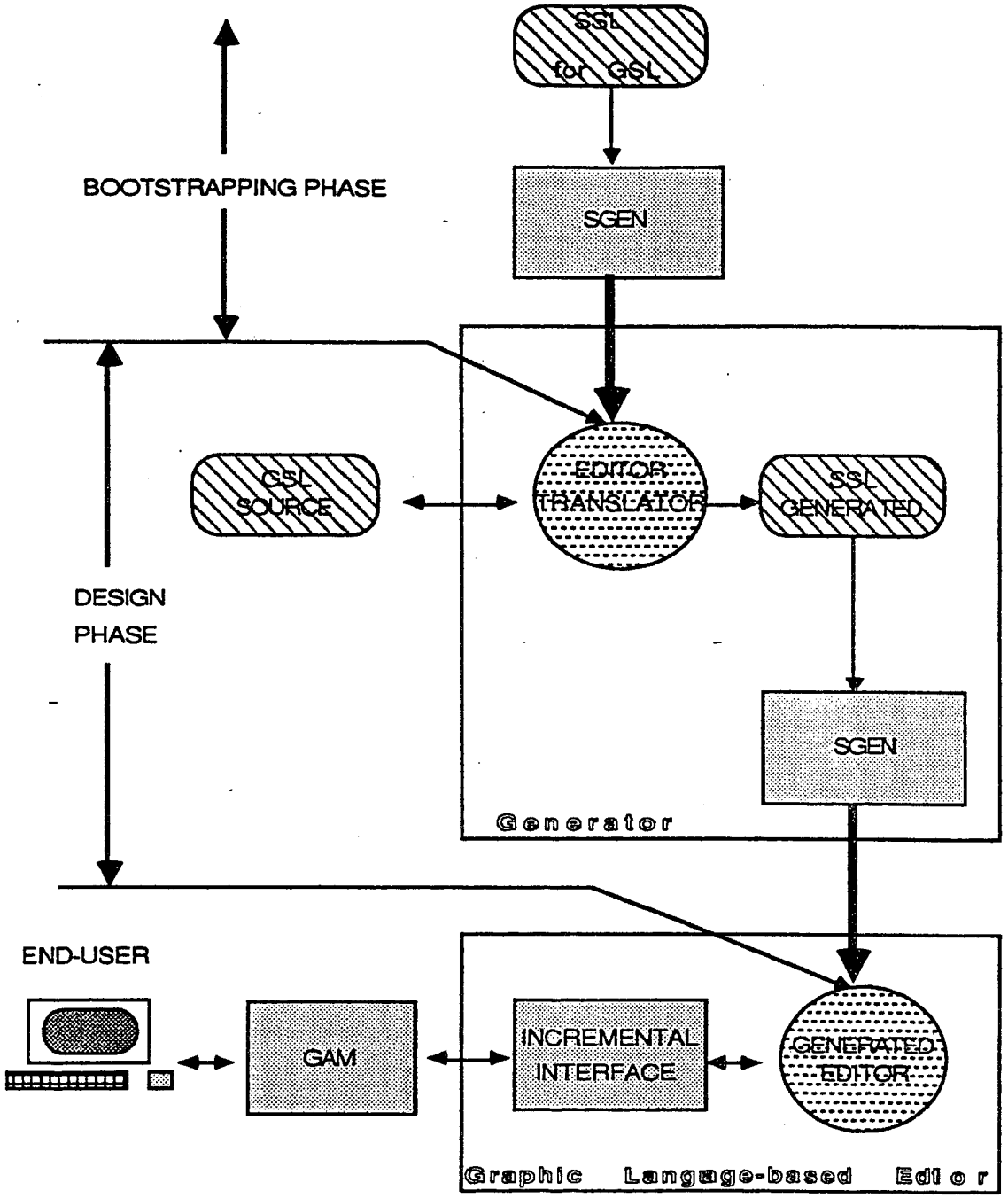


figure 5

Annex B: Equations generated by GSL

We give here the equations generated by the hc,v,vc layout types as shown below:

1. **Horizontal-centered layout: type "hc"**

   - Normal layout
     The box components of such box are placed (w.r.t. their order in the GSL source) from left to right, centered on the reference axis which is the box base-line.

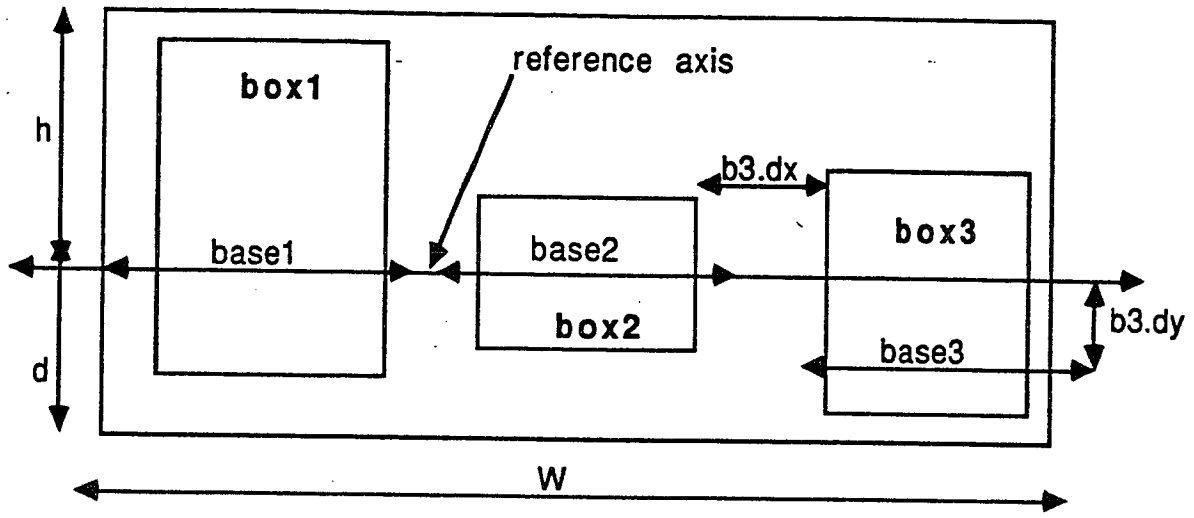


figure 6

   - Synthesized attributes of the father box

[hc1] BOX0.l = sum (BOXk.l + BOXk.dx)
[hc2] BOX0.h = max (0 , max (BOXk.h - BOXk.dy))
[hc3] BOX0.d = max (BOXk.h + BOXk.d + BOXk.dy) - BOX0.h
for k > 0 ;

   - Inherited attributes of the n sub-boxes

[hc4] BOX1.x = BOX1.dx + BOX0.x
" BOXj.x = BOXj-1.x + BOXj-1.l + BOXj.dx
[hc5] BOXk.y = BOX0.h - BOXk.h + BOXk.dy + BOX0.y
for k > 0 and j > 1 ;

2. **Vertical layout: type "v"**

- Normal layout
  The box components of such box are placed (w.r.t. their order in the GSL source) top to downn, along the reference axis which is the left border of the box.

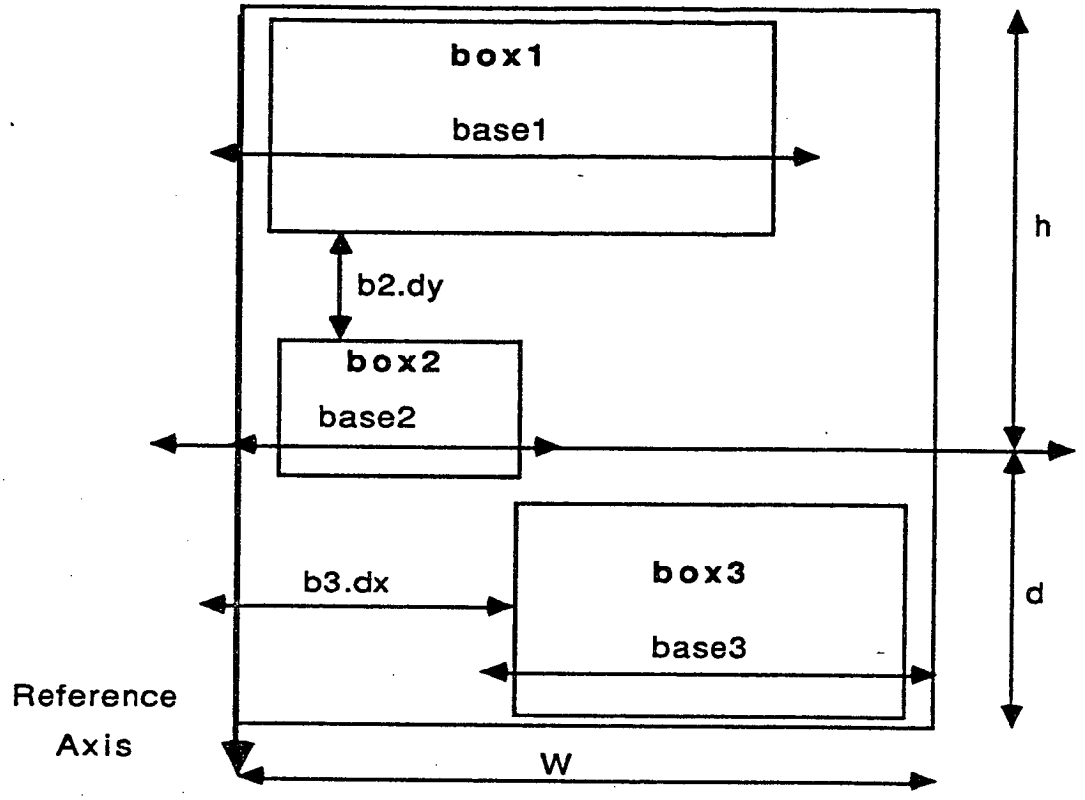


figure 7

- Synthesized attributes of the father box

[v1] BOX0.l = max (BOXk.l + BOXk.dy)
[v2] BOX0.h = sum (BOXk.h + BOXk.d + BOXk.dx)/2
[v3] BOX0.d = BOX0.h
for k > 0 ;

- Inherited attributes of the n sub-boxes

[v4] BOXk.x = BOXk.dx + BOX0.x
[v5] BOX1.y = BOX1.dy + BOX0.y
[v5] BOXj.y = BOXj-1.y + BOXj-1.h + BOXj-1.d + BOXj.dy
for k > 0 and j > 1 ;

3. **Vertical-centered layout: type "vc"**

- Normal layout
  The box components of such box are placed (w.r.t. their order in the GSL source) top to down, centered on the reference axis which is the vertical middle-line of the box.

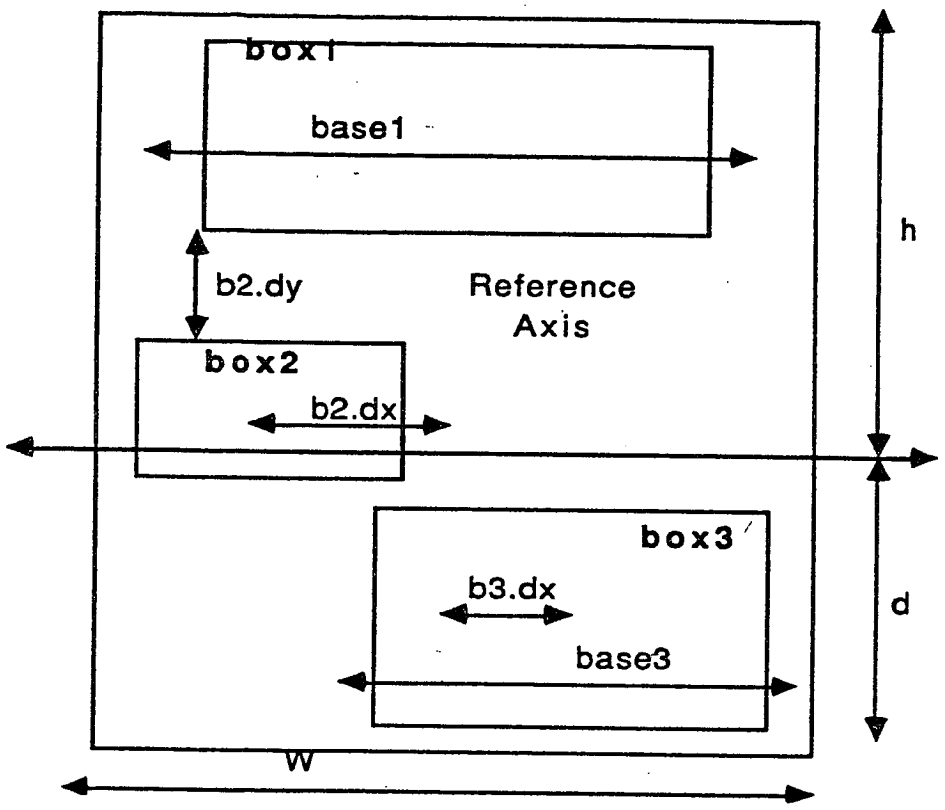


figure 8

- Synthesized attributes of the father box

[vc1] BOX0.l = max(BOXk.l/2 - BOXk.dx) + max(BOXk.l/2 + BOXk.dx)
[vc2] as [v2]
[vc3] as [v3]
for k > 0 ;

- Inherited attributes of the n sub-boxes

[vc4] BOXk.x = max(BOXk.l/2 - BOXk.dx)-BOXk.l/2+BOXk.dx + BOX0.x
[vc5] as [v5]
for k > 0 ;

Annex C: SSL generated for the defined_sum operator

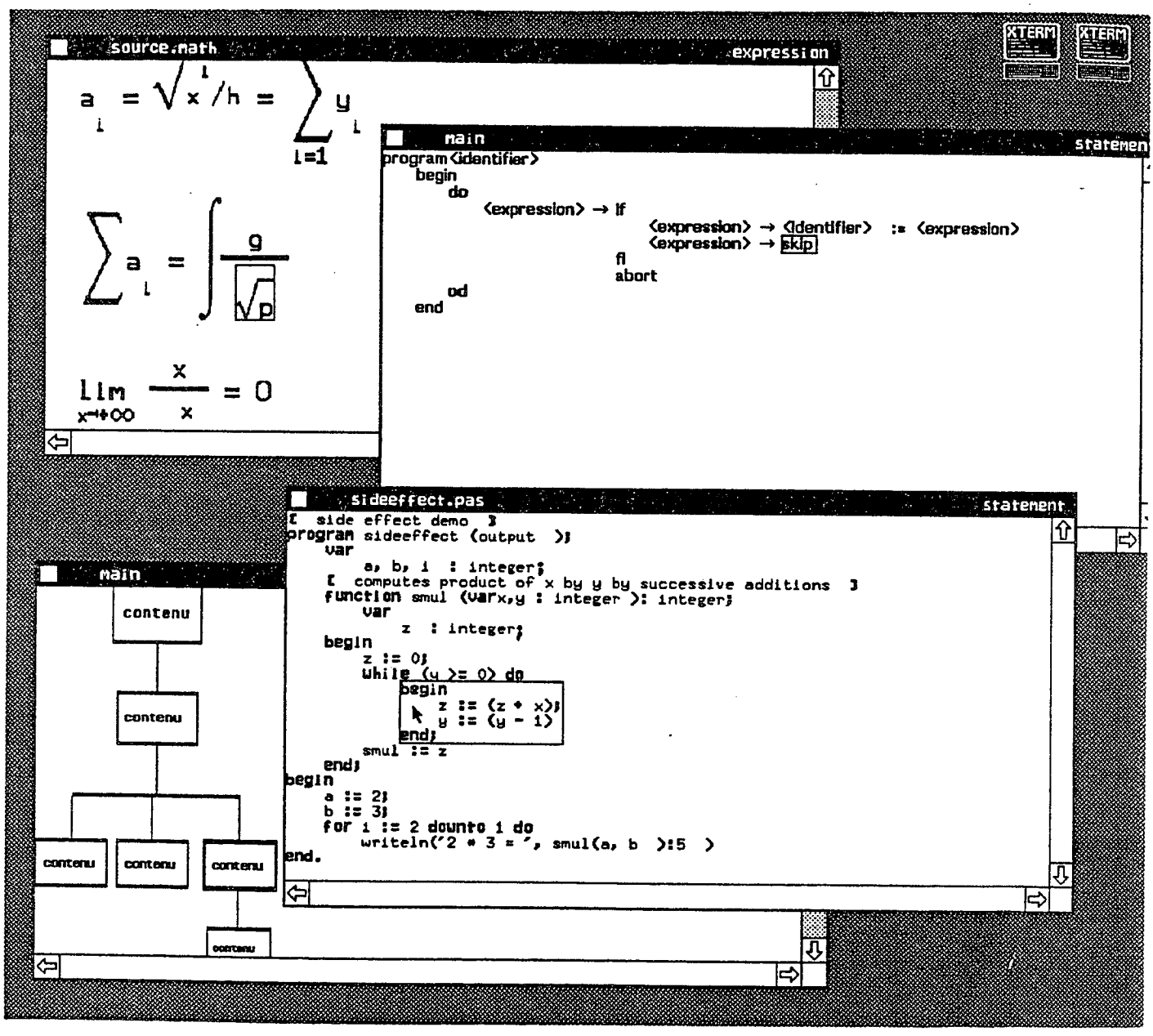
```
/*
*
* Abstract syntax.
* Attributes declarations.
* Attribution rules.
* Unparsing schemes.
*
* Source generated by PREProcessor.
*
*/


/* Attributes declarations */

EXP {
  synthesized REAL h;                    inherited   REAL s;
  synthesized REAL d;                    inherited   REAL x;
  synthesized REAL l;                    inherited   REAL y;
  synthesized REAL H;

};

/* Abstract Syntax */
EXP:
      defined_sum(EXP EXP VAR EXP);

/* Equations       */
EXP:
      defined_sum{
    local REAL Lower_dx;                  local REAL Lower_dy;
    local REAL Var_dx;                    local REAL Var_dy;
    local REAL Diff_dx;                   local REAL Diff_dy;
    local REAL Exp_dx;                    local REAL Exp_dy;
    local REAL Middle_dx;                 local REAL Middle_dy;
    local REAL Upper_dx;                  local REAL Upper_dy;
    local REAL Operands_dx;               local REAL Operands_dy;
    local REAL Symbol_dx;                 local REAL Symbol_dy;
    local REAL defined_sum_dx;            local REAL defined_sum_dy;
    local REAL Diff_h;                    local REAL Diff_d;
    local REAL Diff_l;                    local REAL Diff_H;
    local REAL Diff_s;                    local REAL Diff_x;
    local REAL Diff_y;
    local REAL Middle_h;                  local REAL Middle_d;
```

```
    local REAL Middle_l;                    local REAL Middle_H;
    local REAL Middle_s;                    local REAL Middle_x;
    local REAL Middle_y;
    local REAL Operands_h;                  local REAL Operands_d;
    local REAL Operands_l;                  local REAL Operands_H;
    local REAL Operands_s;                  local REAL Operands_x;
    local REAL Operands_y;
    local REAL Symbol_h;                    local REAL Symbol_d;
    local REAL Symbol_l;                    local REAL Symbol_H;
    local REAL Symbol_s;                    local REAL Symbol_x;
    local REAL Symbol_y;

    defined_sum_dx = 0.0;
    defined_sum_dy = 0.0;
    EXP$1.H = EXP$1.h+EXP$1.d;
    EXP$1.h = MaxReal(0.0, MaxReal(Symbol_h-(Symbol_dy),
 Operands_h-(Operands_dy)));
    EXP$1.d = MaxReal(Symbol_d+Symbol_dy, Operands_d+Operands_dy);
    Symbol_x = 0.0+Symbol_dx;
    Operands_x = Symbol_x+Symbol_l+Operands_dx;
    Symbol_y = EXP$1.h+0.0-(Symbol_h)+Symbol_dy;
    Operands_y = EXP$1.h+0.0-(Operands_h)+Operands_dy;
    EXP$1.l = Symbol_l+Symbol_dx+Operands_l+Operands_dx;
    Symbol_dx = 0.0;
    Symbol_dy = 0.0;
    Symbol_s = EXP$1.s;
    Symbol_H = Symbol_h+Symbol_d;
    Symbol_h = Operands_h-(EXP$2.h);
    Symbol_d = Symbol_h;
    Symbol_l = ((Symbol_h+Symbol_d))/(2.8);
    Operands_dx = 0.0;
    Operands_dy = 0.0;
    Operands_s = EXP$1.s;
    Operands_H = Operands_h+Operands_d;
    Operands_h = EXP$2.d+EXP$2.h+Middle_dy+Middle_h;
    Operands_d = EXP$2.h+EXP$2.d+Upper_dy+Middle_h+Middle_d+
Middle_dy+EXP$4.h+EXP$4.d+Lower_dy-(Operands_h);
    EXP$2.x = Operands_x+Upper_dx;
    Middle_x = Operands_x+Middle_dx;
    EXP$4.x = Operands_x+Lower_dx;
    EXP$2.y = Operands_y+Upper_dy;
    Middle_y = EXP$2.y+EXP$2.h+EXP$2.d+Middle_dy;
    EXP$4.y = Middle_y+Middle_h+Middle_d+Lower_dy;
    Operands_l = MaxReal(EXP$2.l+Upper_dx,
MaxReal(Middle_l+Middle_dx, EXP$4.l+Lower_dx));
```

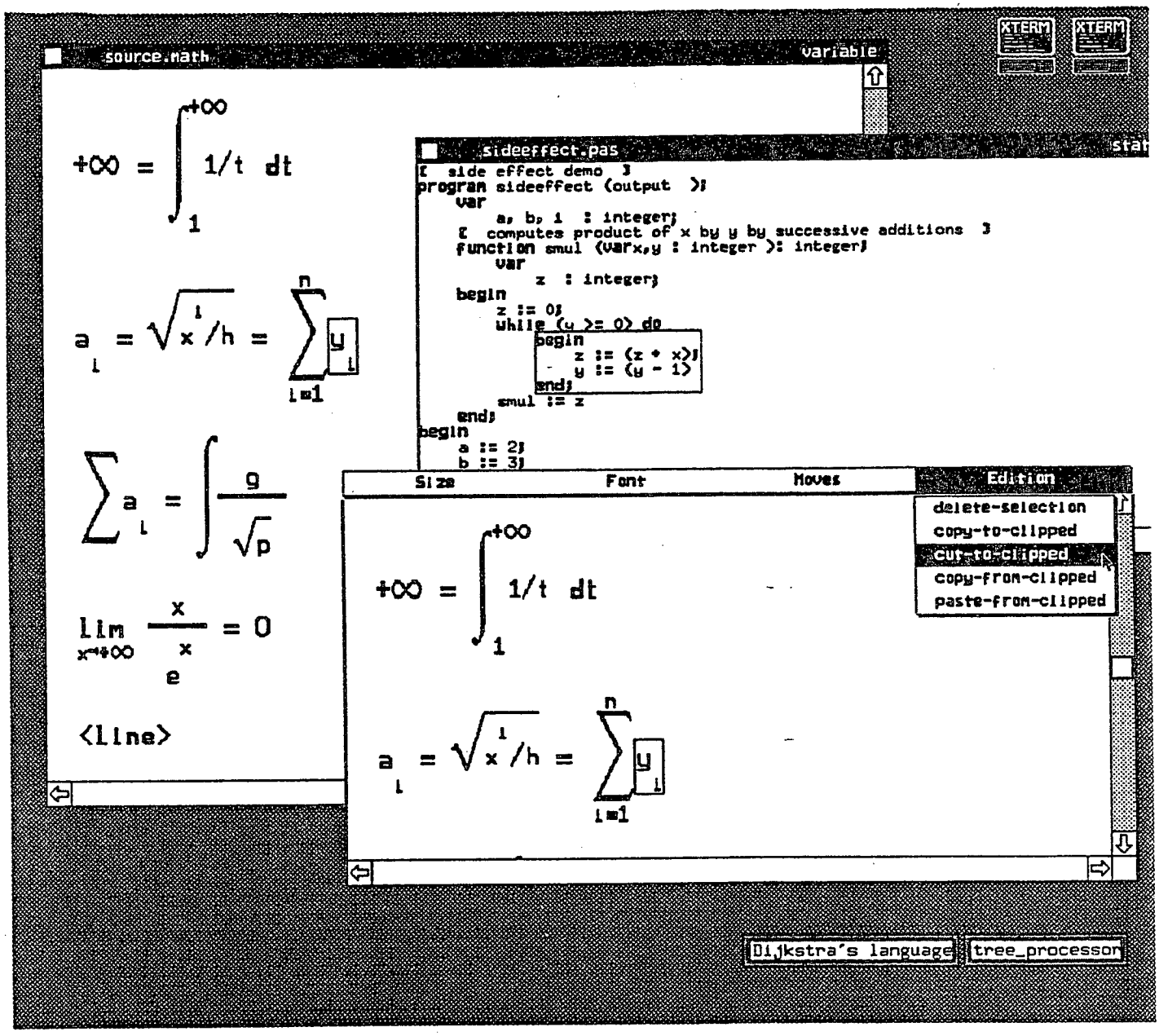
```
    Upper_dx = 0.0;
    Upper_dy = 0.0;
    EXP$2.s = (Symbol_s)*(0.8);
    Middle_dx = 1.0;
    Middle_dy = MaxReal(EXP$2.d+Middle_h, EXP$4.h+Middle_d)+
(1.0)*(Middle_s)-(EXP$2.d)-(Middle_h);
    Middle_s = Operands_s;
    Middle_H = Middle_h+Middle_d;
    Middle_h = MaxReal(0.0, MaxReal(EXP$3.h-(Exp_dy),
MaxReal(Diff_h-(Diff_dy), VAR$1.h-(Var_dy))));
    Middle_d = MaxReal(EXP$3.d+Exp_dy,
MaxReal(Diff_d+Diff_dy, VAR$1.d+Var_dy));
    EXP$3.x = Middle_x+Exp_dx;
    Diff_x = EXP$3.x+EXP$3.l+Diff_dx;
    VAR$1.x = Diff_x+Diff_l+Var_dx;
    EXP$3.y = Middle_h+Middle_y-(EXP$3.h)+Exp_dy;
    Diff_y = Middle_h+Middle_y-(Diff_h)+Diff_dy;
    VAR$1.y = Middle_h+Middle_y-(VAR$1.h)+Var_dy;
    Middle_l = EXP$3.l+Exp_dx+Diff_l+Diff_dx+VAR$1.l+Var_dx;
    Exp_dx = 0.0;
    Exp_dy = 0.0;
    EXP$3.s = Middle_s;
    Diff_dx = 0.0;
    Diff_dy = 0.0;
    Diff_s = Middle_s;
    Diff_H = Diff_h+Diff_d;
    Diff_h = (0.5)*(Diff_s);
    Diff_d = (0.5)*(Diff_s);
    Diff_l = (2.000000e+00)*(Diff_s);
    Var_dx = 0.0;
    Var_dy = 0.0;
    VAR$1.s = Middle_s;
    Lower_dx = 0.0;
    Lower_dy = MaxReal(EXP$2.d+Middle_h,EXP$4.h+Middle_d)+(1.0)*(Middle_s)-
 (Middle_d)-(EXP$4.h);
    EXP$4.s = (Symbol_s)*(0.8);
    };

 /* Unparsing scheme to communicate with GAM */
 EXP: defined_sum[^::="\ncomp "EXP$1.x" "EXP$1.y" "EXP$1.l"
 "EXP$1.H"\t""\ngraph
 "Symbol_x" "Symbol_y" "Symbol_l" "Symbol_H" \"EXP\""@@"\ntext
 "Diff_x" "Diff_y" "Diff_l" "Diff_H" 1 0 0 \" d\""@@"\b"];
```

Annex D: Generated environments

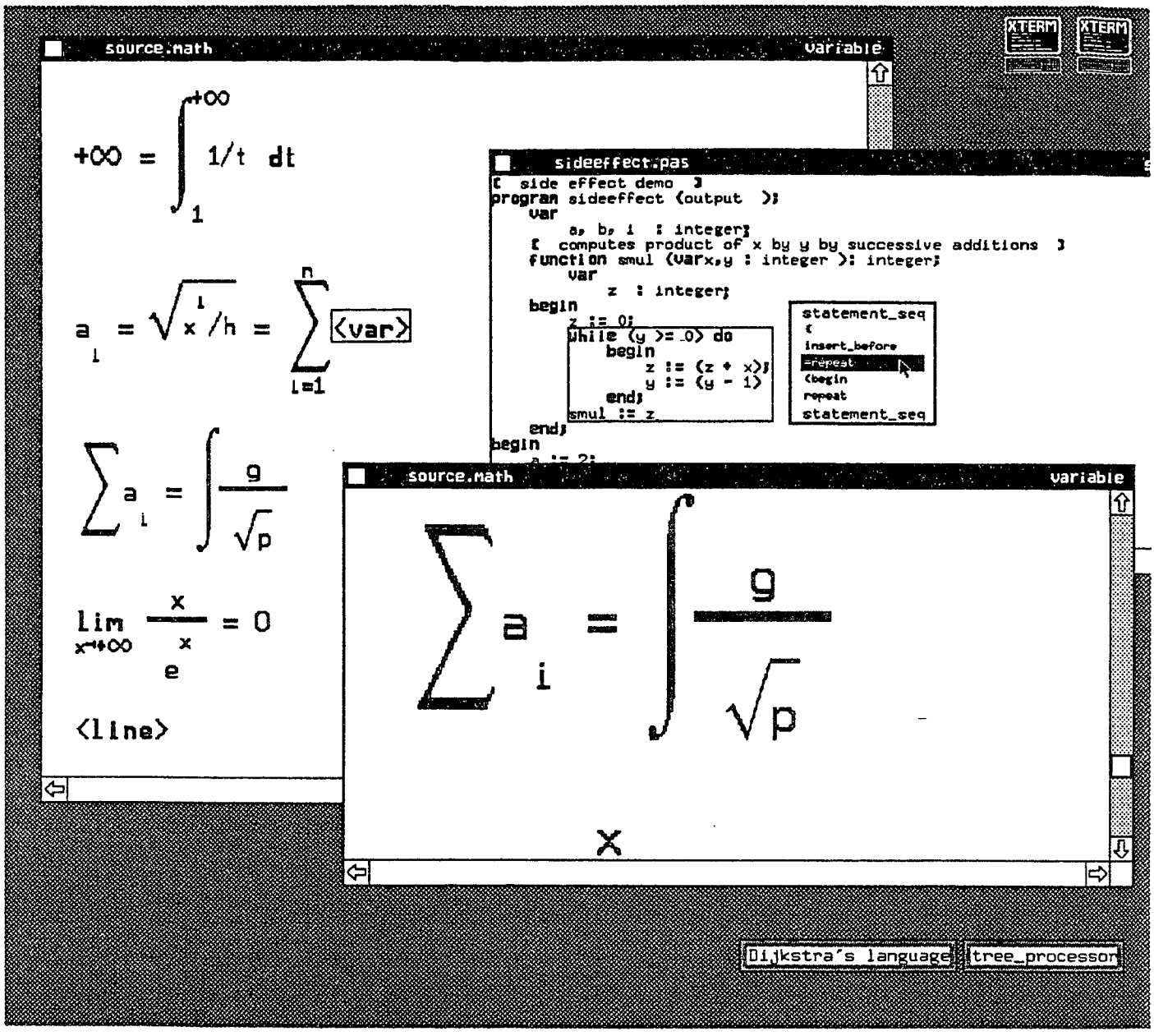
The view below shows a joint session under 4 editors automatically generated by GIGAS: a PASCAL editor, a LISP tree editor, a DIJKSTRA guarded language editor and a mathematical formulae editor. The system provides facilities for cutting, pasting, syntax-oriented menus, transformation menus, zooming, font choosing, etc.



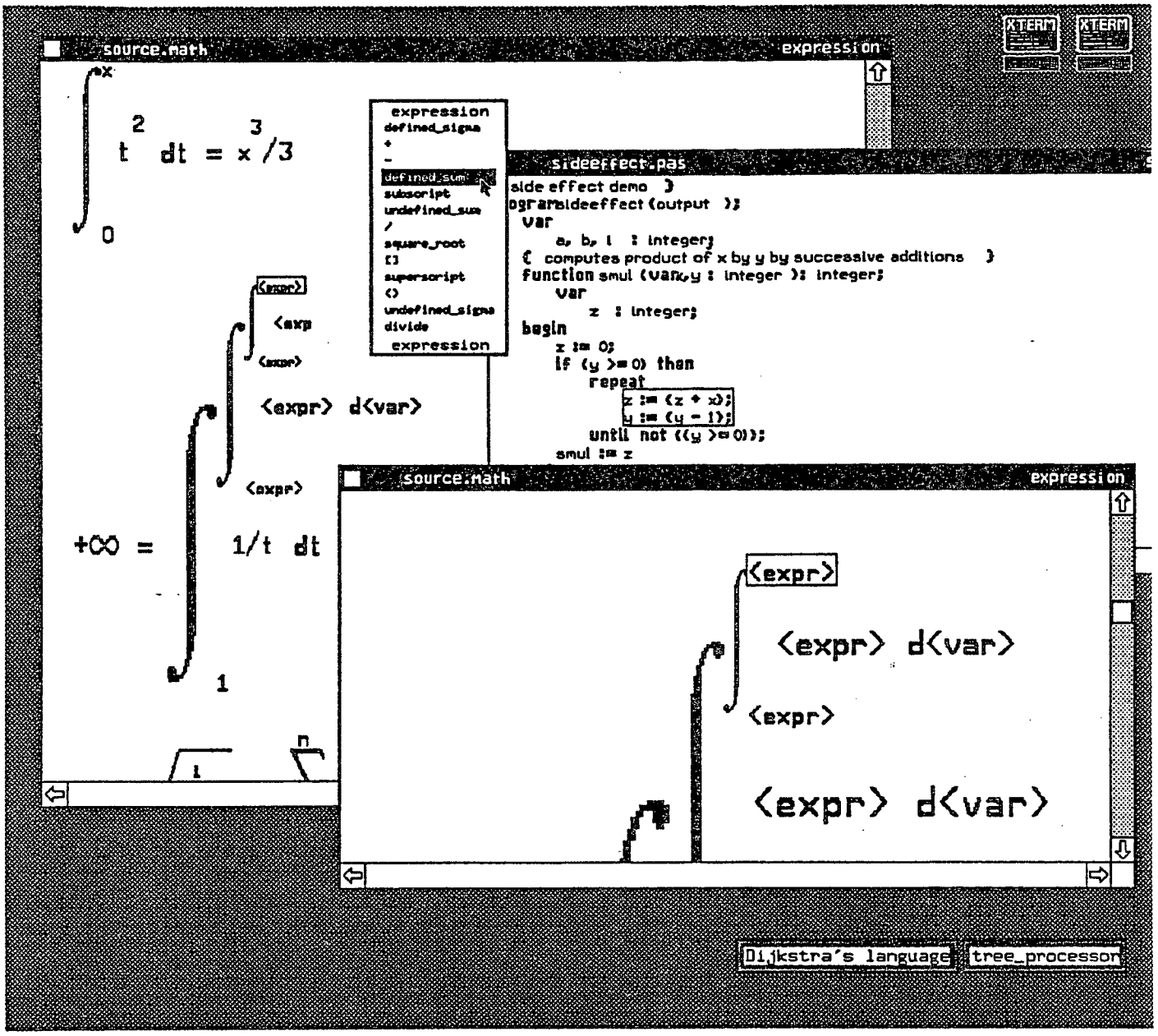
The boxes selected by the mouse are visible; the current Phylum is shown in the right upper corner



Selection of the sigma-operand; cut the subtree using the edition menu



Selection of the PASCAL while; choice of the while-repeat transform in the menu



A menu for expressions and how decreases the size of the nested boxes!

Acknowledgment

We are particularly indebted to X. Ceugniet, B. Chabrier, L. Chauvin, J. M. Deniau, T. Graf, and V. Lextrait for their work on the first implementation of the GIGAS system. Special thanks to I. Attali for helpful comments.

References

[1] Goguen & al. "Initial algebra semantics and continuous algebras" JACM 24, 68-95, 1977

[2] Attali I. & Franchi-Zannettacci P. "Prolog-like schemes for Ada static semantics" Ada UK news, 6, 2, 1985

[3] I. Attali & P. Franchi-Zannettacci," Unification-free execution of TYPOL programs by semantic attribute evaluation" Proc of the 5th Conf on Logic Programming, Seattle, 160-177, 1988

[4] Borras P., Clément D., Despeyroux T., Incerpi J., Kahn G., Lang B., & Pascual V. "CENTAUR: the system" INRIA research report 777, 1987

[5] B. Chabrier, P. Franchi-Zannettacci, V. Lextrait "The GIGAS graphic programming environment generator" Software engineering and its applications, Toulouse, dec 1988

[6] X. Ceugniet, B. Chabrier, L. Chauvin, J.M. Deniau, T. Graf, V. Lextrait, "Prototypage d'un Générateur d'Editeurs Syntaxiques Graphiques, Rapport DESS-ISI, 1987.

[7] Courcelle B. & Franchi-Zannettacci P. "Attribute Grammars and Recursive Program Schemes" TCS 17, vol 1 & 2, 1982

[8] Deransart P., Jourdan M., & Lorho B. "Attribute Grammars: definitions, systems, bibliography" LNCS 323, 1988, Springer Verlag

[9] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang," Programming environnments based on structured editors: The Mentor experience " Rapport INRIA 26, 1980.

[10] A. Demers, T.Reps, T.Teitelbaum, "Incremental evaluation for attribute grammars with application to syntax-directed editors " Conf. Record of 8th ACM Symp.on Principles of Programming Languages, Williamsburg, 105-116.1981

[11] A. Demers, A. Rogers, F.K. Zadeck "Attribute propagation by message passing" ACM Sigplan Symp. on Language issues in programming environements, Seattle, 43-58,1985

[12] A.N.Habermann & D. Notkin "The gandalf Sotware Development Environment" Proc. of the 2nd Symp. on Computation and Information, Monterey, 1983.

[13] M. Jazayeri " A Simpler construction for showing the intrinsically exponential complexity of the circularity problem for attribute grammars" JACM, 28 ,715-720, 1981

[14] G.F. Johnson & C. N. Fischer "A meta-language and system for non local incremental attribute evaluation in language-based editors" Conf. Record of the 12th ACM Symp. on Principles of Programming Languages, New Orleans, 141-151, 1985

[15] M. Jourdan & D. Parigot " The FNC-2 System User's Guide and Reference Manual" version 3.0, INRIA, 1988

[16] Kastens U. "Ordered Attribute Grammars" Acta Informatica 13, 1980

[17] Kennedy K. & Warren S. K. "Automatic generation of efficient evaluators for Attribute Grammars" Proc. of the 3^{rd} ACM Conf on Principle of Programming Languages, Atlanta, 1976

[18] Knuth D. E. "Semantics of Context-Free Languages" Math. Syst. Theory 2, 1968

[19] D.E. Knuth, "TeX et Metafont New directions in Typesetting" Digital Press and the American Society, 1979.

[20] W. R. Mallgreen " Formal specification of interactive graphics programming languages " ACM Distinguished dissertation series, MIT Press, 1983

[21] E. Morcos-Chounet & A. Conchon " PPML: A general formalism to specify prettyprinting " -Proceedings of IFIP Congress, North-Holland, 1986

[22] D. Parigot "Transformations,évaluation incrèmentale et optimisations des grammaires attribuèes: le systéme FNC-2 " Thése Univ. PARIS XI, 1988

[23] T. Reps, C. Marceau T. Teitelbaum " Remote attribute updating for language-based editors " Conf. Record of the13th ACM Symp. on Principles of Programming Languages,St Petersburg, 1-13, 1986

[24] T. Reps & T. Teitelbaum "The Synthesizer Generator Reference Manual" Department of, Compter Science, Cornell University ,1985.

[25] T. Teitelbaum & T. Reps "The Cornell Program Synthesizer: a syntax directed programming environnment" Communications of the ACM, Volume 24 n. 9, 563-573, 1981.

[26] Uhl J., Drossopoulou S., Persch G., Goos G., Dausmann M., & Winterstein G. "An Attribute Grammar for the semantic analysis of Ada" LNCS 149, 1982

[27] P. Van Hentenryck & M Dincbas," Domains in logic programming" AAAI 86, Philadelphia, 1986

[28] W. Waite & G. Goos " Compiler Construction" Springer Verlag 1984

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique